Computer Architecture and Technology Area

Universidad Carlos III de Madrid

OPERATING SYSTEMS

Lab 2. Minishell

BACHELOR'S DEGREE IN COMPUTER SCIENCE AND ENGINEERING

COURSE 2019/2020

Content

1. Lab Statement

This lab allows the student to familiarize with the services for process management that are provided by POSIX. Moreover, one of the objectives is to understand how a Shell works in UNIX/Linux. In summary, a shell allows the user to communicate with the kernel of the Operating System using simple or chained commands.

For the management of processes, you will use the POSIX system calls such as fork, wait, exit. For process communication pipe, dup, close and signal system calls.

The student must design and implement, in C language and over the UNIX/Linux Operating System, a program that acts like a shell. The program must follow strictly the specifications and requirements that are inside this document.

1.1 Lab description

The minishell uses the standard input *(file descriptor = 0)* to read the commands that will be interpreted and executed. It uses the standard output *(file descriptor = 1)* to present the result of the commands on the screen. And it uses the standard error *(file descriptor = 2)* to notify the errors that have happened. If an error occurs in any system call, perror is used to notify it.

1.1.1 Parser

For the development of this lab a 'parser' is given to the student. This parser can read the commands introduced by the user. The student should only work to create a command interpreter. The syntax used by the parser is the following:

A space is a space or a tab.

A separator is a character with a special meaning (| , < , > , &), a new line or the end of file (CTRL-D).

A string is any sequence of characters delimited by a space or a separator.

A command is a sequence of strings separated by spaces. The first string is the name of the command to be executed. The remaining strings are the arguments of the commands. For instance, in the command *ls –l, ls* is the command and *–l* is the argument. The name of the command is to be passed as the argument 0 to the execvp command (*man execvp*). **Each command must execute as an immediate child of the minishell, spawned by fork command** (*man 2 fork*). The value of a command is its termination status (*man 2 wait*), returned by exit function from the child and received by wait function in the father. If the execution fails, the error must be notified by the shell to the user through the standard error.

A command sequence is a list of commands separated by '|'. The standard output of each command is connected through an unnamed pipe to the standard input of the following command. A shell typically waits for the termination of a sequence of commands before requesting the next input line. The value of a sequence is the value returned by the last command in the sequence.

Redirection. The input or the output of a command sequence can be redirected by the following syntax added at the end of the sequence:

- < ***file*** → Use *file* as the standard input after opening it for reading (*man 2 open*).
- > ***file*** → Use *file* as the standard output. If the file does not exist, it is created. If the file exists, it is truncated (*man 2 open/ man creat*).
- !> file → Use file as the standard error. If the file does not exist, it is created. If the file exists, it is truncated (*man 2 open / man creat*).

In case of a redirection error, the execution of the line must be suspended, and the user should be notified using the standard error.

Background (&). A command or a sequence of commands finishing in '&' must execute in background, i.e., the minishell is not blocked waiting for its completion. The minishell must execute the command without waiting and print on the screen the identifier of the child process in the following format:

[%d] \n

The prompt is a message indicating that the shell is ready to accept commands from the user. By default is:

"**MSH>>**"

1.1.2 Command line parsing.

In order to obtain the parsed command line introduced by the user you can use the function *read_command*:

int read_command(char ***argvv, char **filev, int *in_background);

The function returns 0 if the user types Control-C (EOF) and -1 in case of error. **If successful, the function returns the number of commands (*)**. For example:

- For *ls -l* returns 1
- For *ls -l* | *sort* returns 2

(*) The input format must be respected for the correct functioning of the parser: [<command> <args> [| <command> <args>]* [< input_file] [> output_file] [!> outerr_file] [&]

The argument *argvv* contains the commands entered by the user.

The argument *filev* contains the files employed in redirections, if any:

- **filev[0]** contains the file name to be used in standard input redirection and zero if there is no such redirection.
- **filev[1]** contains the file name to be used in standard output redirection and zero if there is no such redirection.
- **filev[2]** contains the file name to be used in standard error redirection and zero if there is no such redirection.

The argument **in_background** is 1 if the command or command sequence are to be executed in background.

Example: If the user enters:

ls -l | sort < fichero

the structure of the arguments of read_command is shown in the following figure:

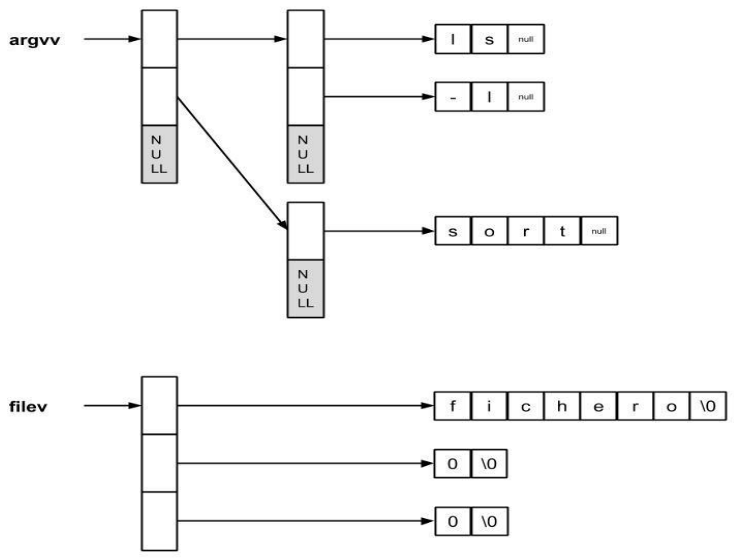

Figura 1: Data structure used by the *parser*.

In the file msh.c (file that must be filled in by the student with the minishell code) the function *read_command* is invoked, and the next loop executed:

```
if (command_counter > 0) {
        if (command_counter > MAX_COMMANDS)
                printf("Error: Numero máximo de comandos es %d \n", MAX_COMMANDS);
        else {
                // Print command
                print_command(argvv, filev, in_background);
        }
}
```

In this code appears the function *print_command()*, which is in charge of printing the stored information captured from the command line, and its code is:

```
for (int i = 0; i < num_commands; i++){
      for (int j = 0; argvv[i][j] != NULL; j++){
         printf("%s\n", argvv[i][j]);
      }
   }

   printf("Redir IN: %s\n", filev[0]);
   printf("Redir OUT: %s\n", filev[1]);
   printf("Redir ERR: %s\n", filev[2]);

   if (in_background == 0)
      printf("No Bg\n");
   else
      printf("Bg\n");
```

It is recommended that the students familiarize themselves with the execution of the provided code, before starting to modify it. This can be done by entering different commands and command sequences and understanding how they are internally handled by the code.

1.2 Development

To develop the minishell we recommend following the next steps, so that it is implemented incrementally. Each step will add a new functionality.

- Execution of simple commands such as *ls -l, who,* etc.
- Execution of simple commands in background.
- Execution of simple commands with redirection (input, output and error).
- Execution of sequences of commands connected through pipes. The number of commands is limited to 3, e.g. *ls –l* | *sort* | *wc*. The implementation of a version that accepts an arbitrary number of commands will be considered for additional marks.
- Execution of simple commands and sequence of commands with redirections (input, output and error), in background (see Appendix to learn about commands n foreground and background to see more details about the requirements of commands in background).
- Execution of internal commands. An internal command is a command, which maps directly to a system call or a command internally implemented inside the shell. It must be implemented and executed inside the minishell (in the parent process). If it finds any error (not enough arguments or other type of error), a notification will appear (using standard error output). The internal commands to be implemented are:

1.2.1 Internal command: mycalc

It works as a simple calculator in the command line. It takes a simple equation, following the format ***operand operator operand***, where operand is an integer number and operator can be sum (add) or module (mod) in which the integer quotient as well as the remainder must be computed.

For the sum operation, the values will be stored in an **environment variable** called "Acc". This variable will start with the value 0, and later the results of the sums will be added (not the results from the module operation). For the module operation, the results will be shown in the following way: **Dividend = Divisor * Quotient + Remainder.**

If the operation is successful, the command will show (**using the standard error output**), the result of solving the computation preceded by the label [OK]. In the case of the sum operation, the accumulated value must be also shown.

If the operator does not correspond with the ones previously described, or not all the terms of the equation were introduced, the following message will be shown (**in the standard output**): "**[ERROR] The structure of the command is <operand 1> <add/mod> <operand 2>"**

Here we show some examples of the usage of this command:

```
msh> mycalc 3 add -8
[OK] 3 + -8 = -5; Acc -5
msh> mycalc 5 add 13
[OK] 5 + 13 = 18; Acc 13
msh> mycalc 10 mod 7
[OK] 10 % 7 = 7 * 1 + 3
msh> mycalc 10 % 7
[ERROR] The structure of the command is <operand 1> <add/mod> <operand 2>
msh> mycalc 8 mas
[ERROR] The structure of the command is <operand 1> <add/mod> <operand 2>
```

1.2.2 Internal command: mycp

It works similarly to the command 'cp'. It takes a source file and a destination and tries to copy the source file to the specified destination, having the same name as the source file.

If any of the source/destination cannot be opened, the following message will be printed (**in the standard output**): “**[ERROR] Error opening original file**” or “**[ERROR] Error opening the copied file**”, depending on the error.

If the function is not called with two parameters, it will show (**using the standard output**) the following message: “**[ERROR] The structure of the command is mycp <original file> <copied file>**”.

If the copy is successful it will show (**using the standard output**): “**[OK] Copy has been successful between <original> and <copied>”.**

Here we show some example of its usage:

```
msh> mycp msh.c msh.c_copy
[OK] Copy has been successful between msh.c and msh.c_copy
msh> mycp no_exist.c dest.c
[ERROR] Error opening original file: No such file or directory
msh> mycp orig.txt
[ERROR] The structure of the command is mycp <original file> <copied file>
```

1.3 Supporting code

To facilitate the realization of this lab you have the file *p2_minishell_19-20.zip* which contains supporting code. To extract the content, you can execute the following:

unzip p2_minishell_19-20.zip

To extract its content, the directory *p2_minishell_19-20/ is created*, where you must develop the lab. Inside that directory the next files are included:

Makefile
Input file for the *make* tool. **It must NOT be modified**. It serves to recompile automatically only the source code that is modified.

libparser.so
Shared library with the parser functions. It allows to recognize and parser the input commands.

probador_ssoo_p2.sh
Shell script that carries out an auto-correction of the student's code. **It must NOT be modified**. To run the script the students must change the permissions *chmod +x probador_ssoo_p2.sh*, and run it *./corrector_ssoo_p2.sh <zip file>*. At the end the students will see printed on screen the estimated grade of their code.

msh.c
C source file which shows an example on how to use the parser. **This file must be modified to do the lab**. It is recommended that you study the function ***read_command*** to understand the lab. The current version simply implements an echo (print) of the commands given to msh as arguments, which are syntactically correct. This functionality must be removed and substitutes by the lines of code that implement the lab.

1.4 Compilation

In order to compile and run the code, it's mandatory to link the shared library *libparser.so*, which is in charge of providing to the programmer (students) a useful function that extracts the list of commands given as input to the minishell. These commands are saved in a global structure that can be manipulated by the programmer.

1. Create a directory in your preferred folder and remember the *path*.
2. Unzip the initial code in *path*.
3. Compile or clean:
    a) With the command *make* you can compile your code.
    b) With the command *make clean* the generated files will be deleted.
4. Run the minishell:
    a) If there is an error because the library is not found:

export LD_LIBRARY_PATH=/home/**username**/**path**:$LD_LIBRARY_PATH

b) Run the minishell again.

2. Assignment submission

2.1 Deadline and method

The deadline for the delivery of this programming assignment in AULA GLOBAL will be **on March 28th (until 23:55h).**

2.2 Submission

The submission must be done using Aula Global through similar links to those used in the first assignment. The submission must be done separately for the code and using **TURNITIN** for the report.

2.3 Files to be submitted

You must submit the code in a zip compressed file with name ssoo_p2_AAAAA_BBBBB_CCCCC.zip where A…A, B…B and C...C are the student identification numbers of the group. A maximum of 3 persons is allowed per group, if the assignment has a single author, the file must be named ssoo_p2_AAAAAAAAA.zip. The file to be submitted must contain:

- msh.c
- Authors.txt: csv file with one author per line with the following format: NIA, Surname, Name

The report must be submitted in a PDF file through TURNITIN. Notice that only PDF files will be reviewed and marked. The file must be named **ssoo_p2_AAAA_BBBBB_CCCCC.pdf**. A minimum report must contain:

- **Cover** with the authors (including the complete name, NIA, and email address).
- **Table of contents**
- **Description of the code** detailing the main functions it is composed of. Do not include any source code in the report.
- **Tests cases** used and the obtained results. All test cases must be accompanied by a description with the motivation behind the tests. In this respect, there are three clarifications to consider.
    - Avoid duplicated tests that target the same code paths with equivalent input parameters.
    - Passing a single test does guarantee the maximum marks. This section will be marked according to the level of test coverage of each program, nor the number of tests per program.

  - Compiling without warnings does not guarantee that the program fulfills the requirements.
- **Conclusions**, describing the main problems found and how they have been solved. Additionally, you can include any personal conclusions from the realization of this assignment.

Additionally, marks will be given attending to the quality of the report. Consider that a minimum report:

- Must contain a title page with the name of the authors and their student identification numbers.
- Must contain an index.
- Every page except the title page must be numbered.
- Text must be justified.

<u>The PDF file must be submitted using the TURNITIN link.</u> Do not neglect the quality of the report as it is a significant part of the grade of each assignment.

<u>NOTE:</u> Only the last version of the submitted files will be reviewed.

3. Rules

1) **Programs that do not compile or do not satisfy the requirements will receive a mark of zero.**
2) **All programs should compile without reporting any warnings.**
3) **Programs without comments will receive a grade of 0.**
4) **The assignment must be submitted using the available links in Aula Global. Submitting the assignments by mail is not allowed without prior authorization.**
5) **The programs implemented must work in the computers of the informatics lab in the university or the Guernika (guernika.lab.inf.uc3m.es) platform. It is the student responsibility to be sure that the delivered code works correctly in those places.**
6) **It is mandatory to follow the input and output formats indicated in each program implemented. In case this is not fulfilled there will be a penalization to the mark obtained.**
7) **It is mandatory to implement error handling methods in each of the programs.**
8) **Students are expected to submit original work. In case plagiarism is detected between two assignments both groups will fail the continuous evaluation. Additional administrative charges of academic misconduct may be filled.**

Failing to follow these rules will be translated into zero marks in the affected programs.

4. Appendix

4.1 Manual (man command).

man is a command that formats and displays the online manual pages of the different commands, libraries and functions of the operating system. If a *section* is specified, man only shows information about name in that section. Syntax:

```
$ man [section] name
```

A man page includes the synopsis, the description, the return values, example usage, bug information, etc. about a *name*. The utilization of man is recommended for the realization of all lab assignments. **To exit a man page, press *q*.**

The most common ways of using man are:

1. **man section element:** It presents the element page available in the section of the manual.
2. **man –a element:** It presents, sequentially, all the element pages available in the manual. Between page and page you can decide whether to jump to the next or get out of the pager completely.
3. **man –k keyword** It searches the keyword in the brief descriptions and manual pages and present the ones that coincide.

4.2 Background and foreground mode

When a simple command is executed in background, the *pid* printed is the one from the process executing that command.

When a command sequence is executed in background, the *pid* printed is the one from the process that executes the last command of the sequence.

With the background operation, is possible that the minishell process shows the prompt mixed with the output of the process child. This is a correct behavior.

After executing a command in foreground, the minishell can not have zombie processes of previous commands executed in background.

4.3 Internal commands

The internal commands (mycalc and mycp) are executed in the minishell process and therefore:

- **They are not part of the command sequences**.
- **They do not have file redirections**.

- **They are not executed in background**.

Bibliography

- C Programming Language (2nd Edition).Brian W. Kernighan , Dennis M. Ritchie.
- The UNIX System S.R. Bourne Addison-Wesley, 1983.
- Advanced UNIX Programming M.J. Rochkind Prentice-Hall, 1985.
- **Operating System Concepts 8th Edition.** Abraham Silberschatz, Yale University, ISBN: 978-0-470-23399-3.
- Programming Utilities and Libraries SUN Microsystems, 1990.
- Unix